
MASSACHUSETTS
INSTITUTE
OF
TECHNOLOGY

TOWARDS A METRICS SUITE FOR OBJECT ORIENTED DESIGN

Shyam Chidamber
Chris F. Kemerer

June 1991

CISR WP No. 226
Sloan WP No. 3323-91-MSA

Accepted for publication in the sixth annual ACM conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA), October 1991.

Center for Information Systems Research
Sloan School of Management
Massachusetts Institute of Technology

TOWARDS A METRICS SUITE FOR OBJECT ORIENTED DESIGN

Abstract

This paper presents theoretical work that builds a suite of metrics for object-oriented design. In particular, these metrics are based upon measurement theory and are also informed by the insights of experienced object-oriented software developers. In evaluating these metrics against a standard set of criteria, they are found to both (a) perform relatively well, and (b) suggest some ways in which the object oriented approach may differ in terms of desirable or necessary design features from more traditional approaches.

In order for object-oriented software production to fulfill its promise in moving software development and maintenance from the current 'craft' environment into something more closely resembling conventional engineering, it will require metrics of the process to aid the software management, project planning and project evaluation functions. While software metrics are a generally desirable feature in any software environment, they are of special importance in the object-oriented approach, since it represents a non-trivial technological change for the organization.

The metrics presented in this paper are the first steps in a project aimed at measuring and evaluating the use of object oriented design principles in organizations.

Accepted for publication in the sixth annual ACM conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA), October 1991.

I. INTRODUCTION

In order for object-oriented software production to fulfill its promise in moving software development and maintenance from the current 'craft' environment into something more closely resembling conventional engineering, it will require measures or *metrics* of the process. While software metrics are a generally desirable feature in the software management functions of project planning and project evaluation, they are of special importance with a new technology such as the object-oriented approach.

This is due to the significant need to train current and new software engineers in generally accepted object-oriented principles. This paper presents theoretical work that builds a suite of metrics for object-oriented design (OOD). In particular, these metrics are based upon *measurement theory* and are informed by the insights of experienced object-oriented software developers. The proposed metrics are evaluated against a widely-accepted list of seven software metric evaluation criteria, and the formal results of this evaluation are presented.

Development and validation of software metrics is expected to provide a number of practical benefits. In general, techniques that provide measures of the size and of the complexity of a software system can be used to aid management in:

- estimating the cost and schedule of future projects,
- evaluating the productivity impacts of new tools and techniques,
- establishing productivity trends over time,
- improving software quality,
- forecasting future staffing needs, and
- anticipating and reducing future maintenance requirements.

More specifically, given the relative newness of the OO approach, metrics oriented towards OO can aid in evaluating the degree of object orientation of an implementation as a learning tool for staff members who are new to the approach. In addition, they may also eventually be useful objective criteria in setting design standards for an organization.

This paper is organized as follows. Section II presents a very brief summary of the need for research in this area. Section III describes the theory underlying the approach taken. Section IV presents the proposed metrics, and Section V presents Weyuker's list of software metric evaluation criteria [Weyuker, 1988]. Section VI contains the results of the evaluation of the proposed metrics, and some concluding remarks are presented in Section VII.

II. RESEARCH PROBLEM

There are two types of criticisms that can be applied to current software metrics. The first category are those criticisms that are leveled at conventional software metrics as they are applied to conventional, non-OO software design and development. These metrics are generally criticized as being without solid theoretical bases[1] and failing to display what might be termed normal predictable behavior [Weyuker, 1988].

The second category is more specific to OO design and development. The OO approach centers around modeling the real world in terms of its objects, which is in stark contrast to older, more traditional approaches that emphasize a function-oriented view that separated data and procedures. Given the fundamentally different notions inherent in these two views, it is not surprising to find that software metrics developed with traditional methods in mind do not readily lend themselves to notions such as classes, inheritance, encapsulation and message passing. Therefore, given that current software metrics are subject to some general criticism and are easily seen as not supporting key OO concepts, it seems appropriate to develop a set, or suite of new metrics especially designed to measure unique aspects of the OO approach.

Some early work has recognized the shortcomings of existing metrics and the need for new metrics especially designed for OO. Some proposals are set out by Morris, although they are empirically suggested rather than theoretically driven [1988]. Pfleeger also suggests the need for new measures, and uses counts of objects and methods to develop and test a cost estimation model for OO development [Pfleeger, 1989; Pfleeger and Palmer, 1990]. Moreau and Dominick suggest three metrics for OO graphical information systems, but do not provide formal, testable definitions [1989]. In contrast, Lieberherr and his colleagues present a well-articulated, formal approach in documenting the Law of Demeter™ [1988] The Demeter system represents a formal attempt at defining the rules of correct object oriented programming style, building on concepts of coupling and cohesion that are used in traditional programming.

Given the extant software metrics literature, the approach taken here is to develop theoretically-driven metrics that can be shown to offer desirable properties, and then choose the most promising candidates for future empirical study. This paper is an initial presentation of six candidate metrics specifically developed for measuring elements contributing to the size and complexity of object-oriented design. Since object design is considered to be a unique aspect of OOD, the proposed metrics directly address this task. The metrics are constructed with a firm basis in theoretical concepts in measurement, while capturing empirical notions of software complexity.

---

[1]For example, see [Vessey and Weber, 1984] and [Kearney, *et al.*, 1986].

III. THEORY BASED METRICS FOR OOD

Booch (1986) defines object oriented design to be the process of identifying objects and their attributes, identifying operations required on each object and establishing interfaces between objects. Design of classes involves three steps: 1) definition of objects, 2) attributes of objects and 3) communication between objects. Methods design involves defining procedures which implement the attributes and operations suffered by objects. Class design is therefore at a higher level of abstraction than the traditional data/procedures approach (which is closer to methods design). It is the task of class design that makes OOD different than data/procedure design [Taylor & Hecht, 1990]. The reader is referred to works by Deutsch, *et al.* [1983], Meyer [1988], Page, *et al.* [1989], Parnas, *et al.* [1986], Seidewitz and Stark [1986] and others for an introduction to the fundamental concepts and terminology of object-oriented design.

Figure 1 shows the fundamental elements of object oriented design as outlined by Booch [1986].

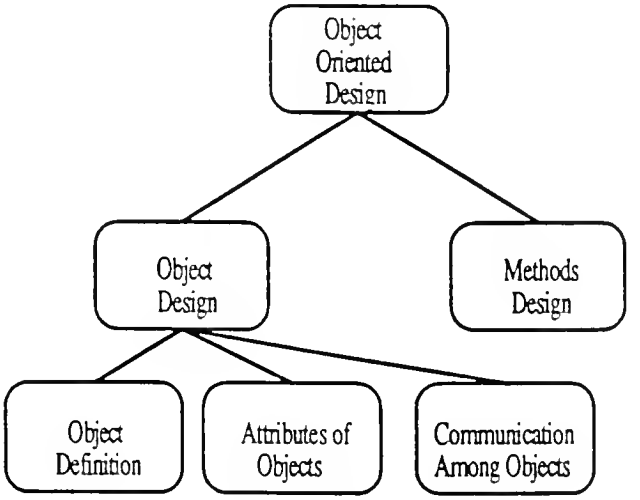


Figure 1: Elements of Object Oriented Design

Measurement theory base

A design can be viewed as a relational system, consisting of object elements, empirical relations and binary operations that can be performed on the object elements.

Notationally: design $P \equiv (A, R_1 \ldots R_n, O_1 \ldots O_m)$

where

A is a set of object elements

$R_1 \ldots R_n$ are *empirical* relations on object elements A (e.g, bigger than, smaller than, etc)

$O_1 ... O_m$ are *binary* operations (e.g., concatenation)

A useful way to conceptualize empirical relations on a set of object elements in this context is to consider the measurement of complexity. A designer has ideas about the complexity of different objects, as to which object is more complex than another. This idea is defined as a *viewpoint*. The notion of a viewpoint was originally introduced to describe evaluation measures for information retrieval systems and is applied here to capture designer views [Chemiavsky, 1971]. An empirical relation is the embodiment of a viewpoint.

A viewpoint is a binary relation $.\geq$ defined on the set P. For P, P', P" $\in$ set P , the following axioms must hold:

$P .\geq P$ (reflexivity)
$P .\geq P'$ or $P' .\geq P$ (completeness)
$P.\geq P', P'.\geq P'' \Rightarrow P.\geq P''$ (transitivity)

i.e., a viewpoint must be of weak order [Zuse, 1987].

To be able to measure something about a object design, the *empirical relational system* as defined above needs to be transformed to a *formal relational system* [Roberts, 1979]. Therefore, let a formal relational system Q be defined as follows:

$Q \equiv (C, S_1 ... S_n, b_1 ... b_m)$
C is a set of elements (e.g., real numbers)
$S_1 ... S_n$ are *formal* relations on C (e.g., >, <, =)
$b_1 ... b_m$ are *binary* operations (e.g., +,-,*)

This is accomplished by a metric μ which maps an empirical system P to a formal system Q. For every element $a \in P$, $\mu(a) \in Q$.

Definitions

The ontological basis principles proposed by Bunge in his "Treatise on Basic Philosophy" forms the basis of the concept of objects [Bunge, 1977]. Consistent with this ontology, objects are defined independent of implementation considerations and encompass the notions of encapsulation, independence and inheritance. According to this ontology, the world is viewed as composed of things, referred to as *substantial individuals* ,and concepts. The key notion is that substantial

individuals possess *properties*. A property is a feature that a substantial individual possesses inherently. An observer can assign features to an individual, these are attributes and not properties. All substantial individuals possess a finite set of properties. "There are no bare individuals except in our imagination" [Bunge, 1979].

Some of the attributes of an individual will reflect its properties. Indeed, properties are recognized only through attributes. A known property must have at least one attribute representing it. Properties do not exist on their own but are "attached" to individuals. On the other hand, individuals are not bundles of properties. A substantial individual and its properties collectively constitute an *object* [Wand, 1987; Wand and Weber, 1990].

An object can be represented in the following manner:

$X = \langle x, p(x) \rangle$ where x is the substantial individual and p(x) is the finite collection of its properties.

x can be considered to be the token or name by which the individual is represented in a system. In object oriented terminology, the instance variables together with its methods are the properties of the object [Banerjee, *et al.*, 1987].

Coupling

Two things are coupled if and only if at least one of them "acts upon" the other [Wand, 1990]. X is said to act upon Y if the history of Y is affected by X, where history is defined as the chronologically ordered states that a thing traverses in time.

let $X = \langle x, p(x) \rangle$ and $Y = \langle y, p(y) \rangle$ be two objects.

$p(x) = \{ S_X \} \cup \{ I_X \}$
$p(y) = \{ S_Y \} \cup \{ I_Y \}$

where $\{ S_i \}$ is the set of methods and $\{ I_i \}$ is the set of instance variables of object *i*.
Using the above definition of coupling, any action by $\{S_X\}$ on $\{S_Y\}$ or $\{I_Y\}$ constitutes coupling, as does any action by $\{S_Y\}$ on $\{S_X\}$ or $\{I_Y\}$. Therefore, any evidence of a method of one object using methods or instance variables of another object constitutes coupling. This is consistent with the law of Demeter™ [Lieberherr, *et al.*, 1988]. In order to promote encapsulation of objects it is generally considered good practice to reduce coupling between objects.

Cohesion

Bunge [1977] defines *similarity* σ() of two objects to be the intersection of the sets of properties of the two objects:

$\sigma(X,Y) = p(x) \cap p(y)$

Following this general principle of defining similarity in terms of sets, the degree of similarity of the methods within the object can be defined to be the intersection of the sets of instance variables that are used by the methods. It should be clearly understood that instance variables are not properties of methods, but it makes intuitive sense that methods that operate on the same instance variables have some degree of similarity.

$\sigma(M_1,M_2...M_n) = \{ M_1 \} \cap \{ M_2 \} \cap \{ M_3 \} ... \{ M_n \}$

where σ() = degree of similarity and $\{ M_i \}$ = set of instance variables used by method M_i.

The degree of similarity of methods relates both to the conventional notion of cohesion in software engineering, (i.e., keeping related things together) as well as encapsulation of objects, that is, the bundling of methods and instance variables in an object. Cohesion of methods can be defined to be the *degree of similarity* of methods. The higher the degree of similarity of methods, the greater the cohesiveness of the methods and the higher the degree of encapsulation of the object.

Complexity of an object

Bunge defines complexity of an individual to be the "numerosity of its composition", implying that a complex individual has a large number of properties. Using this definition as a base, the complexity of an object can be defined to be the cardinality of its set of properties.

Complexity of $<x, p(x)> = | p(x) |$, where $| p(x) |$ is the cardinality of p(x).

Scope of Properties

The scope of a property P in J (a set of objects) is the subset G (P; J) of objects possessing the property.

$G(P; J) = \{ x \mid x \in J \text{ and } P \in p(x) \}$, where p(x) is the set of all properties of all $x \in J$.

Wand defines a class on the basis of the notion of scope [1987]. A class P with respect to a property set p is the set of all objects possessing all properties in p.

$C(p; J) = \cap_{\text{all } P} \{ G(P) \mid P \in p(x) \}$

The inheritance hierarchy is a tree structure with classes as nodes, leaves and a root. Two useful concepts which relate to the inheritance hierarchy can be defined. They are *depth of inheritance* of a class and the *number of children* of a class.

Depth of Inheritance = height of the class in the inheritance tree

The height of a node of a tree refers to the length of the longest path from the node to the root of the tree.

Number of Children = Number of immediate descendents of the class

Both these concepts relate to the notion of scope of properties. i.e., how far does the influence of a property extend? The number of children and depth of inheritance collectively indicate the genealogy of a class. Depth of inheritance indicates the extent to which the class is influenced by the properties of its ancestors and number of children indicates the potential impact on descendents.

Methods as measures of communication

In the object oriented approach, objects can communicate only through message passing. A message can cause an object to "behave" in a particular manner by invoking a particular method. Methods can be viewed as definitions of responses to possible messages [Banerjee, *et al.*, 1987]. It is reasonable therefore to define a *response set* for an object in the following manner:

Response set of an object ≡ {set of all methods that can be invoked in response to a message to the object}

Note that this set will include methods outside the object as well, since methods within the object may call methods from other objects. The response set will be finite, since the properties of an object are finite and there are a finite number of objects in a design.

IV. THE CANDIDATE METRICS

The candidate metrics outlined in this section were developed over a period of several months. This was done in conjunction with a team of software engineers in an organization which has used OOD in a number of different projects over the past four years. Though the primary development language for all projects at this site was C++, the aim was to propose metrics that are not language specific. The viewpoints presented under each metric reflect the object oriented design experiences of many of the engineers, and are presented here to convey the intuition behind each of the metrics.

Metric 1: Weighted Methods Per Class (WMC)

Definition:

Consider a Class C_1, with methods M_1,... Mn. Let c_1,... c_n be the static complexity of the methods. Then

$$WMC = \sum_{i=1}^{n} c_i.$$

If all static complexities are considered to be unity, WMC = n, the number of methods.

Theoretical basis:

WMC relates directly to the definition of complexity of an object, since methods are properties of objects and complexity of an object is determined by the cardinality of its set of properties. The number of methods is, therefore, a measure of object definition as well as being attributes of an object, since attributes correspond to properties.

Viewpoints:

The number of methods and the complexity of methods involved is an indicator of how much time and effort is required to develop and maintain the object.

The larger the number of methods in an object, the greater the potential impact on children, since children will inherit all the methods defined in the object.

Objects with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

Metric 2: Depth of Inheritance Tree (DIT)

Definition:

Depth of inheritance of the class is the DIT metric for the class.

Theoretical basis:

DIT relates to the notion of scope of properties. DIT is a measure of how many ancestor classes can potentially affect this class.

Viewpoints:

The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex.

Deeper trees constitute greater design complexity, since more methods and classes are involved.

It is useful to have a measure of how deep a particular class is in the hierarchy so that the class can be designed with reuse of inherited methods.

Metric 3: Number of children (NOC)

Definition:

NOC = number of immediate sub-classes subordinated to a class in the class hierarchy.

Theoretical basis:

NOC relates to the notion of scope of properties. It is a measure of how many sub-classes are going to inherit the methods of the parent class.

Viewpoints:

Generally it is better to have depth than breadth in the class hierarchy, since it promotes reuse of methods through inheritance.

It is not good practice for all classes to have a standard number of sub-classes. Classes higher up in the hierarchy should have more sub-classes than classes lower in the hierarchy.

The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.

Metric 4: Coupling between objects (CBO)

Definition:

CBO for a class is a count of the number of non-inheritance related couples with other classes.

Theoretical basis:

CBO relates to the notion that an object is coupled to another object if two objects act upon each other, i.e., methods of one use methods or instance variables of another. This is consistent with traditional definitions of coupling as "measure of the degree of interdependence between modules" [Pressman, 1987].

Viewpoints:

Excessive coupling between objects outside of the inheritance hierarchy is detrimental to modular design and prevents reuse. The more independent an object is, the easier it is to reuse it in another application.

Coupling is not associative, i.e., if A is coupled to B and B is coupled to C, this does not imply that C is coupled to A.

In order to improve modularity and promote encapsulation, inter-object couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult.

A measure of coupling is useful to determine how complex the testing of various parts of a design are likely to be. The higher the inter-object coupling, the more rigorous the testing needs to be.

Metric 5: Response For a Class (RFC)

Definition:

RFC = | RS | where RS is the response set for the class.

Theoretical basis:

The response set for the class can be expressed as:

$RS = \{ M_i \} \cup_{all\ n} \{ R_i \}$

where M_i = all methods in the class

and $\{ R_i \}$ = set of methods called by M_i

The response set is a set of methods available to the object and its cardinality is a measure of the attributes of an object. Since it specifically includes methods called from outside the object, it is also a measure of communication between objects.

Viewpoints:

If a large number of methods can be invoked in response to a message, the testing and debugging of the object becomes more complicated.

The larger the number of methods that can be invoked from an object, the greater the complexity of the object.

The larger the number of possible methods that can be invoked from outside the class, greater the level of understanding required on the part of the tester.

A worst case value for possible responses will assist in appropriate allocation of testing time.

Metric 6: Lack of Cohesion in Methods (LCOM)

Definition:

Consider a Class C_1 with methods $M_1, M_2 \ldots, M_n$. Let $\{I_i\}$ = set of instance variables used by method M_i. There are *n* such sets $\{I_1\}, \ldots \{I_n\}$.

LCOM = The number of disjoint sets formed by the intersection of the *n* sets.

Theoretical basis:

This uses the notion of degree of similarity of methods. The degree of similarity for the methods in class C_1 is given by:

$$\sigma() = \{I_1\} \cap \{I_2\} \ldots \cap \{I_n\}$$

If there are no common instance variables, the degree of similarity is zero. However, this does not distinguish between the case where each of the methods operates on unique sets of instance variables and the case where only one method operates on a unique set of variables. The number of disjoint sets provides a measure for the disparate nature of methods in the class. Fewer disjoint sets implies greater similarity of methods. LCOM is intimately tied to the instance variables and methods of an object, and therefore is a measure of the attributes of an object.

Viewpoints:

Cohesiveness of methods within a class is desirable, since it promotes encapsulation of objects.

Lack of cohesion implies classes should probably be split into two or more sub-classes.

Any measure of disparateness of methods helps identify flaws in the design of classes.

Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Summary

The table below summarizes the six metrics in relation to the elements of OOD shown in figure 1.

Metric	Object Definition	Object Attributes	Object Communication
WMC	✔	✔	
DIT	✔		
NOC	✔		
RFC		✔	✔
CBO			✔
LCOM		✔	

Table 1: Mapping of Metrics to OOD Elements

IV. METRICS EVALUATION PROPERTY LIST

Weyuker has developed a list of desiderata for software metrics, and has evaluated a number of existing software metrics using these properties [Weyuker, 1988]. These properties are repeated below.

Property 1: Non-coarseness

Given an object P and a metric μ another object Q can always be found such that:
$\mu(P) \neq \mu(Q)$.

This implies that every object cannot have the same value for a metric, otherwise it has lost its value as a measurement.

Property 2: Non-uniqueness (notion of equivalence)

There can exist distinct objects P and Q such that $\mu(P) = \mu(Q)$. This implies that two objects can have the same metric value, i.e. the two objects are equally complex.

Property 3: Permutation is significant

There exist objects P and Q such that if P is a permutation of Q (i.e., elements in P are simply a different ordering of the elements of Q) then $\mu(P) \neq \mu(Q)$.

Property 4: Implementation not function is important

Suppose there are two object designs P and Q which perform the same function, this does not imply that $\mu(P) = \mu(Q)$. The intuition behind Property 4 is that even though two object designs perform the same function, the details of the implementation matter in determining the object design's metric.

Property 5: Monotonicity

For all objects P and Q, the following must hold:

$\mu(P) \leq \mu(P+Q)$

$\mu(Q) \leq \mu(P+Q)$

where P + Q implies concatenation of P and Q. This implies that objects are minimally zero, and therefore that the combination of two objects can never be less than either of the component objects.

Property 6: Non-equivalence of interaction

Given $\exists$ P, $\exists$ Q, $\exists$ R,

$\mu(P) = \mu(Q)$ does not imply that $\mu(P+R) = \mu(Q+R)$.

This implies that interaction between P and R can be different than interaction between Q and R.

Property 7: Interaction increases complexity

$\exists$ P and Q such that:

$\mu(P) + \mu(Q) \leq \mu(P+Q)$

The idea is that interaction between objects will tend to increase complexity.

V. RESULTS: PROPERTIES OF THE CANDIDATE METRICS

A design goal for all six metrics is their use in analysis of object oriented designs independent of the programming language.in which the application is written. However, there are some basic assumptions made regarding the distribution of objects, methods and instance variables in the discussions for each of the metric properties.

Assumption 1:

Let X_i = The number of methods in a given class *i*.

Y_i = The number of methods called from a given method *i*.

Z_i = The number of instance variables used by a method *i*.

C_i = The number of couplings between a given object *i* and all other objects.

X_i, Y_i, Z_i, C_i are discrete random variables each characterized by some general distribution function. Further, all the X_is are independent and identically distributed. The same is true for all the Y_is, Z_is and C_is.

Assumption 2: $X_i \geq 1$ i.e., each class contains one or more methods.

Assumption 3: Two classes can have identical methods, in the sense that combination of the two classes into one class would result in one of the methods being redundant.

Assumption 4: The inheritance tree is "full" i.e., there is a root, several intermediate nodes which have siblings, and leaves. The tree is not balanced, i.e., each node does not necessarily have the same number of children.

These assumptions while believed to be reasonable, are of course subject to future empirical test.

Metric 1: Weighted Methods Per Class (WMC)

Let X_p = number of methods in class P and X_q = number of methods in class Q.

Let y = probability $X_p \neq X_q$, and (1 - y) = probability $X_p = X_q$

As $0 < P < 1$ from assumption 1, there is a finite probability that $\exists$ a Q such that $\mu(P) \neq \mu(Q)$, therefore property 1 is satisfied. Similarly, $0 < 1 - y < 1$, there is a finite probability that $\exists$ a Q such that $\mu(P) = \mu(Q)$. Therefore property 2 is satisfied. Permutation of elements inside the object does not alter the number of methods of the object. Therefore Property 3 is not satisfied. The function of the object does not define the number of methods in a class. The choice of methods is an implementation decision, therefore Property 4 is satisfied.

Let $\mu(P) = n_p$ and $\mu(Q) = n_q$, then $\mu(P+Q) = n_p + n_q$. Clearly, $\mu(P+Q) \geq \mu(P)$ and $\mu(P+Q) \geq \mu(Q)$, thereby satisfying property 5. Now, let $\mu(P) = n$, $\mu(Q) = n$, $\exists$ an object R such that it has a number of methods ∂ in common with Q but no methods in common with P. Let $\mu(R) = r$.

$\mu(P+R) = n + r$

$\mu(Q+R) = n + r - \partial$

therefore $\mu(P+Q) \neq \mu(Q+R)$ and property 6 is satisfied. For any two objects P and Q, $\mu(P+Q) = n_p + n_q - \partial$, where n_p is the number of methods in P, n_q is number of methods in Q and P and Q have ∂ methods in common.

Clearly, $n_p + n_q - \partial \leq n_p + n_q$ for all P and Q.

i.e., $\mu(P+Q) \leq \mu(P) + \mu(Q)$ for all P and Q.

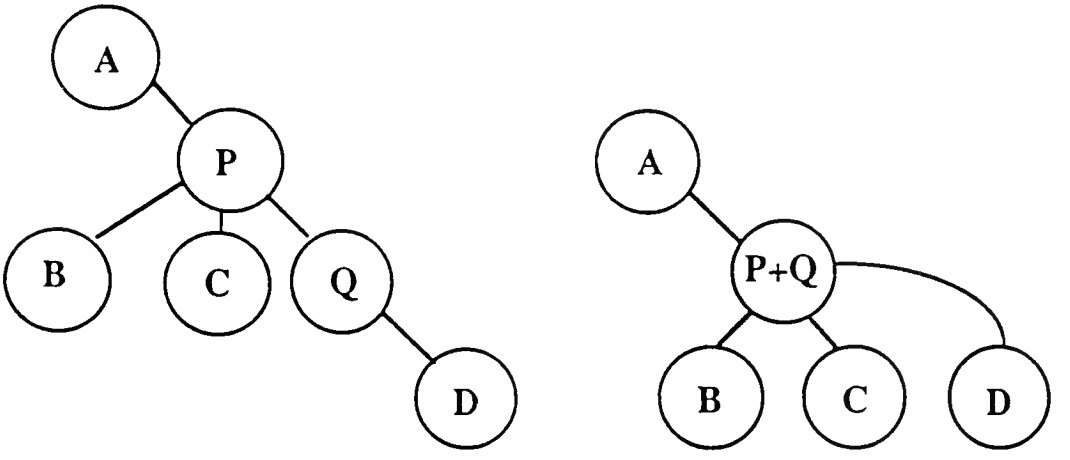
Therefore Property 7 is not satisfied.

Metric 2: Depth of Inheritance Tree (DIT)

Per assumption 4, every tree has a root and leaves. The depth of inheritance of a leaf is always greater than the root. Therefore, property 1 is satisfied. Also, since every tree has at least some nodes with siblings, there will always exist at least two objects with the same depth of inheritance, i.e., property 2 is satisfied. Permutation of the elements within an object does not alter the position of the object in the inheritance tree, and therefore property 3 is not satisfied. Implementation of an object involves choosing what properties the object must inherit in order to perform its function. In other words, depth of inheritance is implementation dependent, and property 4 is satisfied.

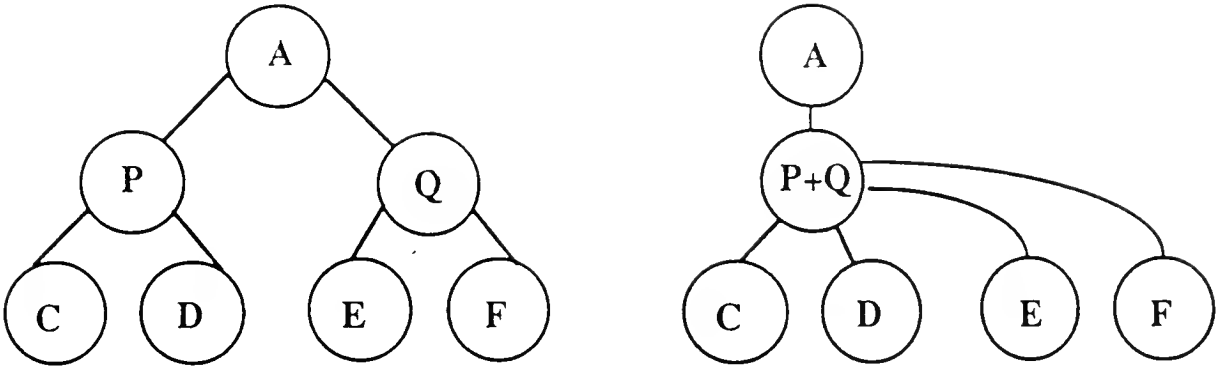
When any two objects P & Q are combined, there are three possible cases:

i) when one is a child of the other:



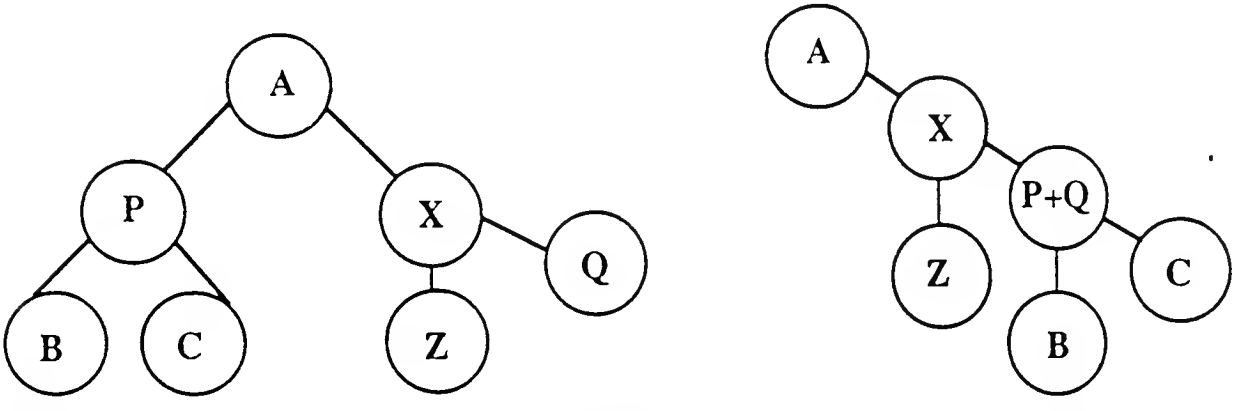
In this case, $\mu(P) = n$, $\mu(Q) = n + 1$, but $\mu(P+Q) = n$, i.e. $\mu(P+Q) < \mu(Q)$. Property 5 is not satisfied.

Case ii) P & Q are siblings



In this case, $\mu(P) = \mu(Q) = n$ and $\mu(P+Q) = n$, i.e. Property 5 is satisfied.

Case iii) P & Q are not directly connected.



If P+Q moves to P's location in the tree, Q does cannot inherit methods from C, however if P+Q moves to Q's location, P maintains its inheritance. Therefore, P+Q will be in Q's old location. In this case, $\mu(P) = x$, $\mu(y)$ and $y > x$. $\mu(P+Q) = y$, i.e., $\mu(P+Q) > \mu(P)$ and $\mu(P+Q) = \mu(Q)$ and property 5 is satisfied. Since $\mu(P+Q) \geq \mu(P)$ is not satisfied for all possible cases, Property 5 is not satisfied. Let P and Q be siblings, i.e. $\mu(P) = \mu(Q) = n$, and let R be a child of P. Then $\mu(P+R) = n$ and $\mu(Q+R) = n + 1$. i.e. $\mu(P+R)$ is not equal to $\mu(Q+R)$. Property 6 is satisfied. For any two objects P & Q, $\mu(P+Q) = \mu(P)$ or $= \mu(Q)$. Therefore, $\mu(P+Q) \leq \mu(P) + \mu(Q)$ i.e. Property 7 is not satisfied.

Metric 3: Number Of Children (NOC)

Let P and R be leaves, $\mu(P) = \mu(R) = 0$, let Q be the root $\mu(Q) > 0$. $\mu(P) \neq \mu(Q)$ therefore property 1 is satisfied. Since $\mu(R) = \mu(P)$, Property 2 is also satisfied. Permutation of elements within an object does not change the number of children of that object, therefore Property 3 is not satisfied. Implementation of an object involves decisions on the scope of the methods declared within the object, i.e, the sub-classing for the object. The number of sub-classes is therefore dependent upon implementation of the object. Therefore, property 4 is satisfied. Let P and Q be two objects with n_p and n_q sub-classes respectively (i.e., $\mu(P) = n_p$ and $\mu(Q) = n_q$). Combining P and Q, will yield a single object with $n_p + n_q - \partial$ sub-classes, where ∂ is the number of children P and Q have in common. Clearly, ∂ is 0 if either n_p or n_q is 0. Now, $n_p + n_q - \partial \geq n_p$ and $n_p + n_q - \partial \geq n_q$. This can be written as:

$\mu(P+Q) \geq \mu(P)$ and $\mu(P+Q) \geq \mu(Q)$ for all P and all Q.

Therefore, Property 5 is satisfied. Let P and Q each have *n* children and R be a child of P which has *r* children. $\mu(P) = n = \mu(Q)$. The object obtained by combining P and R will have (n-1) + r children, whereas an object obtained by combining Q and R will have $n + r$ children, which means that $\mu(P+R) \neq \mu(Q+R)$. Therefore property 6 is satisfied.

Given any two objects P and Q with n_p and n_q children respectively, the following relationship holds:

$\mu(P) = n_p$ and $\mu(Q) = n_q$.

$\mu(P+Q) = n_p + n_q - \partial$

where ∂ is the number of common children.

Therefore, $\mu(P+Q) \leq \mu(P) + \mu(Q)$ for all P and Q. Property 7 is not satisfied.

Metric 4: Response for a Class (RFC)

Let X_p = RFC for class P

X_q = RFC for class Q.

Let y = probability $X_p \neq X_q$, (1 - y) = probability $X_p = X_q$

$X_p = F(Y_i)$ and $X_q = F(Y_j)$ i.e., X_p is some function of the number of methods called by a method in class P. Now, F() is monotonic in Y, since the response set can only increase as the number of methods called increases. Y_i and Y_j are independent identically distributed discrete random variables, as per assumption 1. Therefore, $F(Y_i)$ and $F(Y_j)$ are also discrete random variables that are i.i.d. Therefore, there is a finite probability that $\exists$ a Q such that $\mu(P) \neq \mu(Q)$ resulting in property 1 being satisfied. Also as $0 < 1 - y < 1$ there is a finite probability that $\exists$ a Q such that $\mu(P) = \mu(Q)$, therefore property 2 is satisfied. Permutation of elements within an object does not change the number of methods called by that object, and therefore property 3 is not satisfied. Implementation of an object involves decision about the methods that need to be called

and therefore Property 4 is satisfied. Let P and Q be two classes with RFC of P = n_p and RFC of Q = n_q. If these two classes are combined to form one class, the response for that class will be the larger of the two RFC values for P and Q $\Rightarrow \mu(P+Q) = \text{Max}(n_p, n_q)$. Clearly, $\text{Max}(n_p,n_q) \geq n_p$ and $\text{Max}(n_p,n_q) \geq n_q$ for all possible P and Q. $\mu(P+Q) \geq \mu(P)$ and $\geq \mu(Q)$ for all P and Q. Therefore, property 5 is satisfied. Let P, Q and R be three classes such that, $\mu(P) = \mu(Q) = n$ and $\mu(R) = r$. Then $\mu(P+Q) = \text{Max}(n,r)$ and $\mu(Q+R) + \text{Max}(n,r)$. i.e., $\mu(P+Q) = \mu(R+Q)$. Therefore property 6 is not satisfied. For any two classes P and Q, $\mu(P+Q) = \text{Max}(\mu(P), \mu(Q))$. Clearly, $\text{Max}(\mu(P), \mu(Q)) \leq \mu(P) + \mu(Q)$ which means that Property 7 is not satisfied.

Metric 5: Lack Of Cohesion Of Methods (LCOM)

Let X_p = LCOM for class P

X_q = LCOM for class Q.

Let y = probability $X_p \neq X_q$, (1 - y) = probability $X_p = X_q$

$X_p = F(Y_i)$ and $X_q = F(Y_j)$ i.e., X_p is some function of the number of instance variables used by a method in class P. Now, F() is monotonic in Y, since the LCOM can only decrease as the number of instance variables used increases. Y_i and Y_j are independent identically distributed discrete random variables, as per assumption 1. Therefore, $F(Y_i)$ and $F(Y_j)$ are also discrete random variables that are i.i.d. therefore property 1 is satisfied. Also as $0 < 1 - y < 1$. then there is a finite probability that $\exists$ a Q such that $\mu(P) = \mu(Q)$, therefore property 2 is satisfied. Permutation of the elements of an object does not alter the set of methods called from that object, consequently not changing the value of LCOM. Therefore, property 3 is not satisfied. The LCOM value depends on the construction of methods, which is implementation dependent, making LCOM also implementation dependent and satisfying property 4. Let P and Q be any two objects with $\mu(P) = n_p$ and $\mu(Q) = n_q$. Combining these two objects can potentially reduce the number of disjoint sets. i.e., $\mu(P+Q) = n_p + n_q - \partial$ where ∂ is the number of disjoint sets reduced due to the combination of P and Q. The reduction ∂ is some function of the particular sets of instance variables of the two objects P and Q. Now, $n_p \geq \partial$ and $n_q \geq \partial$ since the reduction in sets obviously cannot be greater than the number of original sets. Therefore, the following result holds:

$n_p + n_q - \partial \geq n_p$ for all P and Q and

$n_p + n_q - \partial \geq n_q$ for all P and Q.

Property 5 is satisfied.

Let P and Q be two objects such that $\mu(P) = \mu(Q) = n$, and let R be another object with $\mu(R) = r$.

$\mu(P+Q) = n + r - \partial$, similarly

$\mu(Q+R) = n + r - ß$

Given that ∂ and ß are not functions of n, they need not be equal. i.e., $\mu(P+R) \neq \mu(Q+R)$, satisfying property 6. For any two objects P and Q, $\mu(P+Q) = n_p + n_q - \partial$. i.e.,

$\mu(P+Q) = \mu(P) + \mu(Q) - \partial$ which implies that

$\mu(P+Q) \leq \mu(P) + \mu(Q)$ for all P and Q.

Therefore property 7 is not satisfied.

Metric 6: Coupling Between Objects (CBO)

As per assumption 1, there exist objects P, Q and R such that $\mu(P) \neq \mu(Q)$ and $\mu(P) = \mu(R)$ satisfying properties 1 and 2. Permutation of the elements inside an object does not change the number of inter-object couples, therefore property 3 is not satisfied. Inter-object coupling occurs when methods of one object use methods or instance variables of another object, i.e., coupling depends on the construction of methods. Therefore property 4 is satisfied. Let P and Q be any two objects with $\mu(P) = n_p$ and $\mu(Q) = n_q$. If P and Q are combined, the resulting object will have $n_p + n_q - \partial$ couples, where ∂ is the number of couples reduced due to the combination. That is $\mu(P+Q) = n_p + n_q - \partial$, where ∂ is some function of the methods of P and Q. Clearly, $n_p - \partial \geq 0$ and $n_q - \partial \geq 0$ since the reduction in couples cannot be greater than the original number of couples. Therefore,

$n_p + n_q - \partial \geq n_p$ for all P and Q and

$n_p + n_q - \partial \geq n_q$ for all P and Q

i.e., $\mu(P+Q) \geq \mu(P)$ and $\mu(P+Q) \geq \mu(Q)$ for all P and Q. Thus, property 5 is satisfied. Let P and Q be two objects such that $\mu(P) = \mu(Q) = n$, and let R be another object with $\mu(R) = r$.

$\mu(P+Q) = n + r - \partial$, similarly

$\mu(Q+R) = n + r -$ ß

Given that ∂ and ß are not functions of n, they need not be equal, i.e., $\mu(P+R)$ is not equal to $\mu(Q+R)$, satisfying property 6. For any two objects P and Q, $\mu(P+Q) = n_p + n_q - \partial$.

$\mu(P+Q) = \mu(P) + \mu(Q) - \partial$ which implies that

$\mu(P+Q) \leq \mu(P) + \mu(Q)$ for all P and Q.

Therefore property 7 is not satisfied.

Summary of results

All six metrics fail to meet property 3, suggesting that perhaps permutation of elements within an object is not significant. The intuition behind this is that measurements on class design should not depend on ordering of elements within it, unlike program bodies where permutation of elements should yield different measurements reflecting the nesting of if-then-else blocks.

The rationale behind property 7 according to Weyuker is to "allow for the possibility of increased complexity due to potential interaction" [Weyuker, 1988]. All six metrics fail to meet this, suggesting that perhaps this is not applicable to object oriented designs. This also raises the issue that complexity could increase, not reduce as a design is broken into more objects. Further research in this area is needed to clarify this issue.

The RFC metric fails to satisfy property 6 and the DIT metric fails to satisfy property 5. These deficiencies are a result of the definition of the two metrics and further refinements will be required to satisfy these properties. It is worth pointing out that Harrison [1988] and Zuse [1991] have criticized the non-equivalence of interaction property (property 6) and note that this property may not be widely applicable. Also, the DIT metric, as shown earlier does not satisfy the monotonicity property (property 5) only in the case of combining two objects in different parts of the tree, which empirical research may demonstrate to be a rare occurrence. Table 2 presents a summary of the metrics properties.

Summary of Results

METRIC	P1	P2	P3	P4	P5	P6	P7
WMC	Yes	Yes	NO	Yes	Yes	Yes	NO
DIT	Yes	Yes	NO	Yes	NO	Yes	NO
NOC	Yes	Yes	NO	Yes	Yes	Yes	NO
RFC	Yes	Yes	NO	Yes	Yes	NO	NO
LCOM	Yes	Yes	NO	Yes	Yes	Yes	NO
CBO	Yes	Yes	NO	Yes	Yes	Yes	NO

Table 2: Summary of Metrics Properties

VI. CONCLUDING REMARKS

This research has developed a new set of software metrics for OO design. These metrics are based in measurement theory, and also reflect the viewpoints of experienced OO software developers. In evaluating these metrics against a set of standard criteria, they are found to both (a) perform relatively well, and (b) suggest some ways in which the OO approach may differ in terms of desirable or necessary design features from more traditional approaches. Clearly some future

research designed both to extend the current proposed metric set and to further investigate these apparent differences seems warranted.

In particular, this set of six proposed metrics is presented as a first attempt at development of formal metrics for OOD. They are unlikely to be comprehensive, and further work could result in additions, changes and possible deletions from this suite. However, at a minimum, this proposal should lay the groundwork for a formal language with which to describe metrics for OOD. In addition, these metrics may also serve as a generalized solution for other researchers to rely on when seeking to develop specialized metrics for particular purposes or customized environments.

Currently planned empirical research will attempt to validate these candidate metrics by measuring them on actual systems. In particular, a three-phased approach is planned. In Phase I, the metrics will be measured on a single pilot system. After this pilot test, Phase II will consist of calculating the metrics for multiple systems and simultaneously collecting some previously established metrics for purposes of comparison. These previously existing metrics could include such well-known measures as source lines of code, function points, cyclomatic complexity, software science metrics, and fan-in/fan-out. Finally, Phase III of the research will involve collecting performance data on multiple projects in order to determine the relative efficacy of these metrics in predicting managerially relevant performance indicators.

It is often noted that OO may hold some of the solutions to the software crisis. Further research in moving OO development management towards a strong theoretical base should provide a basis for significant future progress.

REFERENCES

Abbot, R. J. (1987). "Knowledge Abstraction," *Communications of the ACM*, 30, 664-671.

Banerjee, J., *et al.* (1987). "Data Model Issues for Object Oriented Applications," *ACM Transactions on Office Information Systems*, 5, January, 3-26.

Booch, G. (1986). "Object Oriented Development," *IEEE Transactions on Software Engineering*, SE-12, February, 211-221.

Bunge, M. (1977). *Treatise on Basic Philosophy : Ontology I : The Furniture of the World.* Boston, Riedel.

Bunge, M. (1979). *Treatise on Basic Philosophy : Ontology II : The World of Systems.* Boston, Riedel.

Cherniavsky, V. and D. G. Lakhuty (1971). "On The Problem of Information System Evaluation," *Automatic Documentation and Mathematical Linguistics*, 4, 9-26.

Cunningham, W. and K. Beck (1987). "Constructing Abstractions for Object Oriented Applications", Computer Research Laboratory, Textronix Inc. Technical Report CR-87-25, 1987.

Deutsch, P. and A. Schiffman (1983). "An Efficient Implementation of the Smalltalk-80 System," *Conference record of the Tenth Annual ACM Symposium on the Principles of Programming Languages.*

Fenton, N. and A. Melton (1990). "Deriving Structurally Based Software Measures," *Journal of Systems and Software*, 12, 177-187.

Harrison, W. (1988). "Software Science and Weyuker's Fifth Property", University of Portland Computer Science Department Internal Report 1988.

Hecht, A. and D. Taylor (1990). "Using CASE for Object Oriented Design with C++," *Computer Language*, 7, November, Miller Freeman Publications, San Francisco, CA.

Kearney, J. K., *et al.* (1986). "Software Complexity Measurement," *Communications of the ACM*, 29 (11), 1044-1050.

Lieberherr, K., *et al.* (1988). "Object Oriented Programming : An Objective Sense of Style," *Third Annual ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*. 323-334.

Meyer, B. (1988). *Object Oriented Software Construction (Series in Computer Science)*. New York, Prentice Hall International.

Moreau, D. R. and W. D. Dominick (1989). "Object Oriented Graphical Information Systems: Research Plan and Evaluation Metrics," *Journal of Systems and Software*, 10, 23-28.

Morris, K., (1988). *Metrics for Object Oriented Software Development*, unpublished Masters Thesis, M.I.T., Cambridge, MA.

Page, T., *et al.* (1989). "An Object Oriented Modelling Environment," *Proceedings of the Fourth Annual ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA)*.

Parnas, D. L., *et al.* (1986). Enhancing Reusability with Information Hiding. *Tutorial: Software Reusability*. P. Freeman, Ed., New York, IEEE Press. 83-90.

Peterson, G. E. (1987). *Tutorial: Object Oriented Computing*. IEEE Computer Society Press.

Pfleeger, S. L. (1989). "A Model of Cost and Productivity for Object Oriented Development", Contel Technology Center Technical Report.

Pfleeger, S. L. and J. D. Palmer (1990). "Software Estimation for Object Oriented Systems," *Fall International Function Point Users Group Conference*. San Antonio, Texas, October 1-4, 181-196.

Pressman, R. S. (1987). *Software Engineering: A Practioner's Approach*. New York, McGraw Hill.

Roberts, F. (1979). *Encyclopedia of Mathematics and its Applications*. Addison Wellesley Publishing Company.

Seidewitz, E. and M. Stark (1986). "Towards a General Object Oriented Software Development Methodology," *First International Conference on the ADA Programming Language Applications for the NASA Space Station*. D.4.6.1-D4.6.14.

Vessey, I. and R. Weber (1984). "Research on Structured Programming: An Empiricist's Evaluation," *IEEE Transactions on Software Engineering*, SE-10 (4), 394-407.

Wand , Y. and R. Weber (1990). "An Ontological Model of an Information System," *IEEE Transactions on Software Engineering*, 16, N 11, November, 1282-1292.

Wand, Y. (1987). A Proposal for a Formal Model of Objects. *Research Directions in Object Oriented Programming*. Ed., Cambridge, MA, M.I.T. Press. 537-559.

Weyuker, E. (1988). "Evaluating Software Complexity Measures," *IEEE Transactions on Software Engineering*, 14, No 9, September, 1357-1365.

Wybolt, N. (1990). "Experiences with C++ and Object Oriented Software Development," *USENIX Association C++ Conference Proceedings*. San Francisco, CA.

Zuse, H. (1991). *Software Complexity: Measures and Methods*. New York, Walter de Grutyer.

Zuse, H. and P. Bollman (1987). "Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics", I.B.M. Research Center Technical Report RC 13504, August.